


1. [Introduction](#)
2. [Background](#)
3. [Motivation](#)
4. [Implementation](#)
5. [Results](#)
6. [Conclusions](#)
7. [The Team](#)

## Introduction

An introduction to the D/A conversion design and process.


## What are D/A and A/D Converters

Digital to Analog (D/A) and Analog to Digital (A/D) converters are pervasive and essential in the technology of today. Televisions, smart phones, radars, and even sensors in your car all require quality conversion in order to work effectively. The improvements and characteristics of these converters have significantly supported the digital revolution in past decades. D/A and A/D converters are intrinsically inversely related. Digital to analog converters are systems that take discrete digital data and convert it into continuous analog signals, whereas analog to digital converters achieve the opposite effect. Specifically, we will be investigating and implementing a D/A converter that takes stored digital data and reproduces the original audio signal. An important concept to note is that digital signals can undergo manipulation and storage without damaging or losing the original data because of their discrete nature. For our project, we chose an R/2R implementation of a D/A converter using a BeagleBone Black. We will further explore these new concepts later in this educational document, but here is our poster which gives a surface-level explanation on most of the topics.



# R/2R Implementation of a D/A Converter

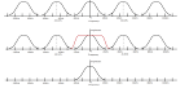
Eric Miller, Tyler Taldone, Paul Rockaway, Tam Vu  
lovetf.elecs@rice.edu



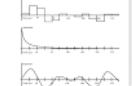
### Digital to Analog Conversion

- D/A converts discrete digital data into continuous analog signals
- Accuracy, cost, resolution, speed, size, and power consumption characterize most D/A converters
- We chose a binary weighted D/A design for its practicality

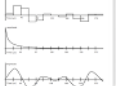
Frequency Domain



The D/A Process

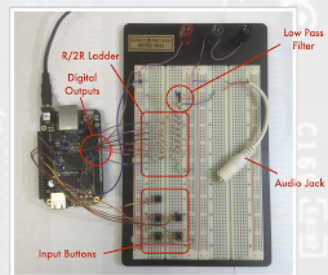


Time Domain




- To produce an analog signal from digital values:
  - Produce the infinite frequency quantized signal from the digital data
  - Remove higher frequency copies of the signal using a low pass filter
  - Output the result which is a reproduction of the original analog signal

### The Design



### Full Digital Signal

- The BeagleBone Black is an embedded computer running a Linux OS
- BeagleBone Black outputs 8 new bits 44,100 times per second




B0

B1

...

B7

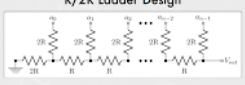


- Changes 8 output bits to a 0 or a 1 based on the audio file we read
- Yields 256 total voltage levels of output

### R/2R Ladder Output


- Feed these 8 outputs in a resistor ladder with R and 2R resistors
- Voltage at output becomes the weighted sum of each bit

### R/2R Ladder Design



- Quantization levels make the signal appear blocky
- Introduces higher frequencies to the signal


### Digital Sine Wave



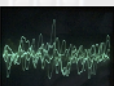
### Audio Output

- Remove higher frequencies with a low pass filter
- Filter out frequencies above 20,000 Hz to keep all audible sounds
- Final result is the original audio signal reproduced
- Play back at 44.1 kHz to reproduce any desired sound

Reproduced Sine Wave



Beethoven's 5th



### Conclusions

- Implementation Results**
  - Cost: made of simple components
  - Power consumption: completely passive implementation
  - Size: needs small number of components
  - Speed: achieves 44.1 kHz reproduction rate
- Implementation Shortcomings**
  - Accuracy: Linux multitasker randomly delays the signal
  - Resolution: low bit count causes quantization error
- Future Improvements**
  - Use a real time operating system to remove signal delays
  - Up-sample on the fly to reduce noise
  - Increase number of bits to increase resolution

### Acknowledgments

We would like to thank Dr. Ray Sinar for his mentorship and support with this project.  
 Special thanks to Derrick Malloy for his advice, and for providing us a copy of his book *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux*.  
 The BeagleBone Black was provided by the Department of Electrical and Computer Engineering at Rice University.

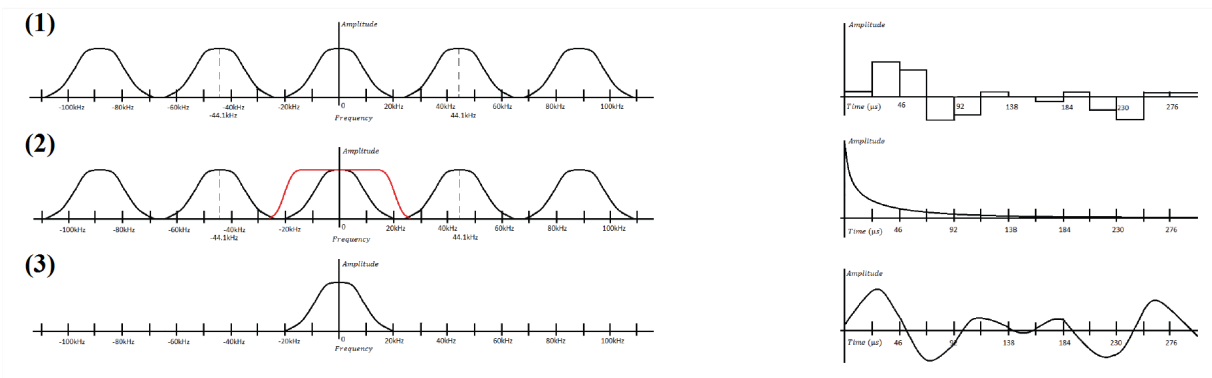
## Characteristics of D/A Converter

There are multiples D/A architecture designs that one can choose to implement. Depending on which architecture one selects, the characteristics of the digital to analog conversion are affected. Some characteristics to keep in mind when designing a digital to analog converter are speed, accuracy, cost, resolution, power consumption, and size. The speed characterizes the frequency of the sample production of the audio signal. Accuracy determines how true the produced signal is compared to the original. Cost is determined by the price of the components used in making the converter and can also be impacted by the power consumption if measured over time. The resolution is the number of voltage output levels which is based on how many bits the converter uses. We chose to have 8 bits in our converter so we had  $2^8 \text{ bits} = 256$  voltage levels. There also exists a reciprocity between speed and resolution because the more bits it uses, the more difficult it is to reproduce them at a high frequency. The power consumption of the circuit

depends on how high these other characteristics are and whether one chose to implement a passive (low consumption) or active (high consumption) system. Active systems include components that control flow and inject power in a circuit, such as transistors, while passive systems have components that cannot amplify the signal, such as resistors. Finally, the size of a D/A converter relies on the scale and number of components used.

## The D/A Process

First, we will discuss the conversion process in terms of systems and signals, and then provide a real world example so that one can see these concepts realized.



Take note that the graphs on the left represent signals in the frequency domain while graphs on the right are signals in the time domain. Typically, when reproducing an analog signal from digital values, one starts with stored discrete data that looks like (1) in the frequency domain when produced into a quantized signal. Looking at the time domain in (1), you can see the quantized nature of the signal. Note that in the example, the original digital signal has nonzero amplitude from -20kHz to 20kHz. This range is because our D/A converter has the purpose of reproducing audio and the human ear can only hear up to the 20kHz range. Anything above this range we cannot hear. However, as you can see there are copies of this original digital signal at higher frequencies. We do not want these high frequency copies because they consume power when outputted, even though we cannot hear them. When experimentally outputting this

unfiltered digital signal, we also found that they also introduced noise and damaged our speaker. So, in order to remove these harmful high frequency copies and isolate the original digital signal that occurs at lower frequencies, we want to apply a low pass filter as seen in (2). In the time domain, we display the impulse response of a simple resistor-capacitor low pass filter to help show this cutoff of higher frequencies. Now, in (3), one can see that the high frequency copies of the original quantized signal have been filtered out. The converter then output this result which is actually an exact reproduction of the original analog signal!

Now, armed with the general D/A conversion process in mind, consider an mp3 player producing a song through a speaker. The song audio is originally a digital stream of bits that is stored in the player's memory. This digital data can be stored into groups, called packets, or just outputted raw into the D/A converter. It takes these bit streams and turns them into a digital signal with distinct quantization levels and high frequency copies. Then, this digitized signal goes through a low pass filter which effectively smoothes out this quantized signal into a continuous analog signal and removes the high frequency copies above 20kHz. This signal is a reproduction of the original analog signal that was recorded by means of a A/D converter! In order to amplify this signal so that we can hear it, the analog signal powers an audio amplifier that in turn powers a speaker which produces the sounds that we hear.

Background

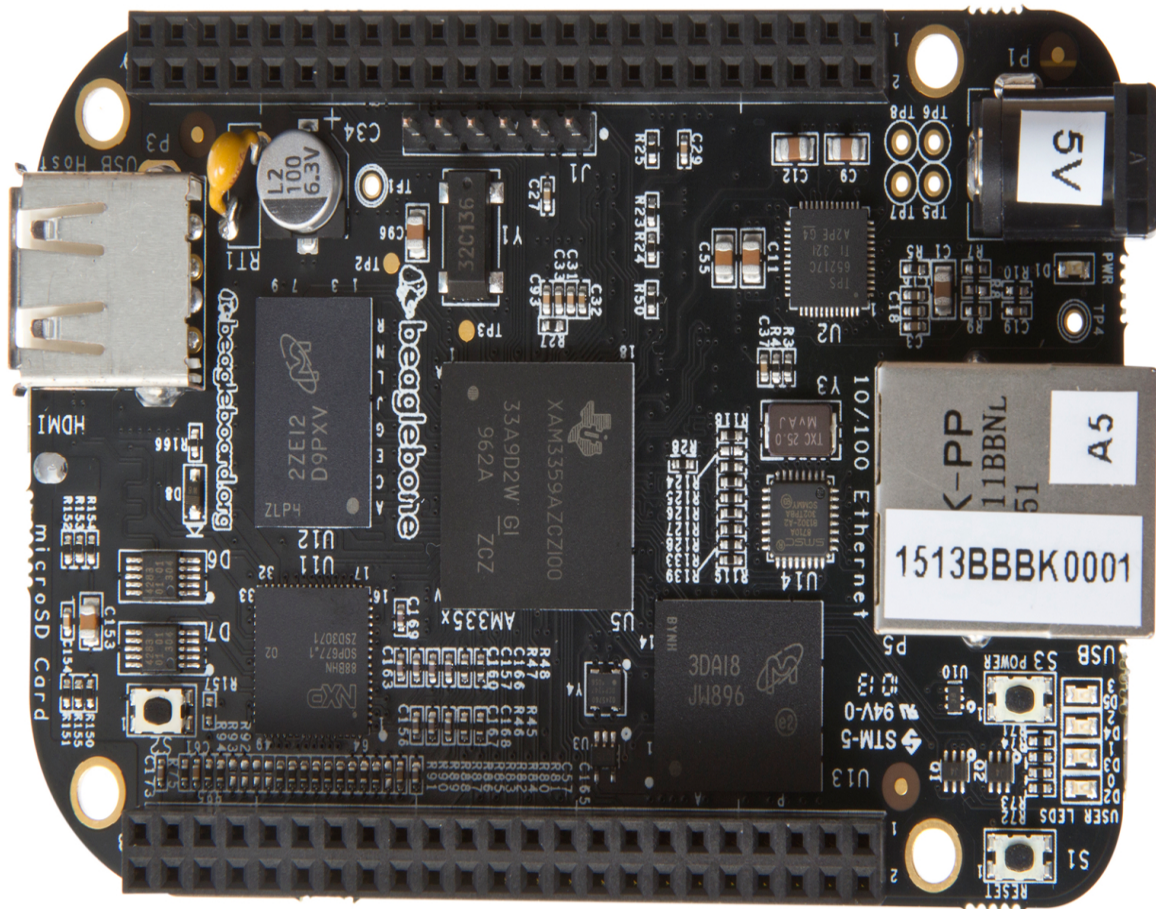
Background of our topic.

## **Current D/A Methods**

Remember that there are many uses for D/A conversion, not just audio reproduction, so different methods are used for different practical applications. One of the most frequently used methods in industry is D/A converters that oversample or interpolate. Oversampling is where one samples a signal at a much higher frequency than is specified by the Nyquist rate. This means that it samples much more often than at twice the bandwidth of a signal. This oversampling usually result in effectively shifting the noise present in the important low frequencies into high frequencies which can then easily be low pass filtered. This results in D/A converters with high resolution and low cost. Another commonly used type, and the one that most closely reflects what we chose in our project, is a binary-weighted D/A converter. This type has more focus on hardware implications, rather than the software implications of the oversampling mentioned earlier. Binary-weighted D/A converters require components that take each of the bits entering the converter to a summing point where they add up to an accurate  $V_{out}$  value. The disadvantage to this method is that the D/A converter is inaccurate due to the individual tolerances of the components. A subcategory of this converter type is an R/2R ladder, that has resistor values of R and 2R for each of the input bit streams. This fixes the accuracy problem for low resolution converters since it is simple to find resistors of these R/2R values.

## **What is a BeagleBone Black?**





The BeagleBone Black is a single-board embedded computer that is made with open source software and hardware for developmental purposes. For our project, we were provided a BeagleBone Black by the Department of Electrical and Computer Engineering at Rice University, but it is much cheaper than a full-fledged computer. Its processing power and storage space are much smaller, but if one plugged in an HDMI to a monitor and a USB to a keyboard, it can seem like a real computer running Linux. There are 46 total I/O pins but many of them are reserved for special uses, so only approximately 30 of them are general purpose I/O. With the processor speed and I/O count in mind, we chose our design to rely on using 8 pins and therefore 8 bits. Also note that the audio files we stored on the BeagleBone Black have exactly a 44.1kHz sampling rate.

## Motivation

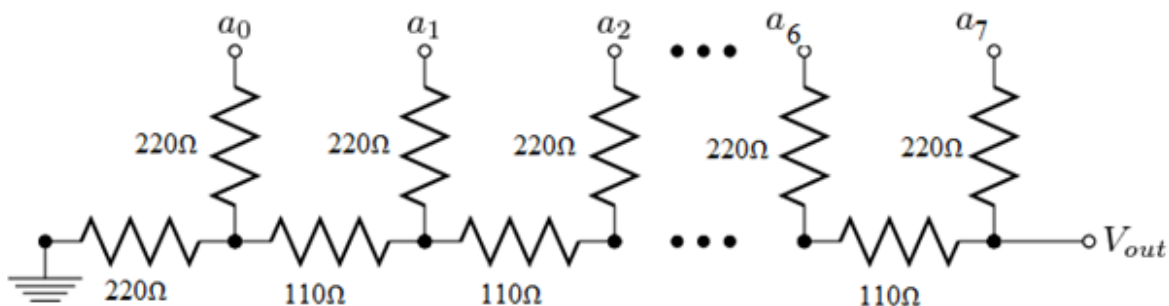
A summary of our motivations and choices for this topic.

## D/A Conversion

We were interested in this project because we wanted to explore the science and practicality behind D/A conversion but also to learn from the hardware/software interactions afforded by the BeagleBone Black. We wanted to take a stored digital signal and take it through the D/A conversion process primarily for the purpose of teaching others and learning ourselves. This process is enormously important in the world of contemporary technology and we are executing a novel process with this specific BeagleBone black. Controlling hardware/software interactions is integral in the design process for many electronics, so we wanted to incorporate this as well.

## Why choose the R/2R Ladder?

Since we are using 8 bits to quantize our signal, an overall effective D/A converter that changes our digital signal to analog is the R/2R ladder. As discussed earlier, an R/2R ladder is a binary-weighted converter that uses resistors of only two different values: R and 2R (the actual values are insignificant, what matters is the 2:1 ratio). These resistors are cascaded together in the structure below, allowing for the output voltage to be a weighted sum of the input voltages.





Although there are other ways to implement a D/A converter, for digital signals of 8 bits or less the R/2R ladder is one of the best options. Some of the advantages of the R/2R ladder is that it is composed of only resistors of two different values, allowing it to be very easily implemented on a small-scale at a low cost (small resistors of specific values are can be cheaply produced). Some potential drawbacks of using an 2/2R ladder is that with longer ladders, the cumulative capacitance of the system could potentially delay the transmission of the signal as it is converted from digital to analog. However, since we are using 8 bit signals, there are only 8 rungs on our ladder, so there is no significant delay time due to the capacitance. Another potential problem with R/2R ladders in general is that with the more significant bits, the precision of the resistors are increasingly important. A small fluctuation in these resistors can completely overwhelm the output values of the smaller bits. Fortunately for us, since there are only 8 rungs in the ladder, only so much precision is required for our resistors in the most significant bit.

### **Why choose RC Low Pass Filter**

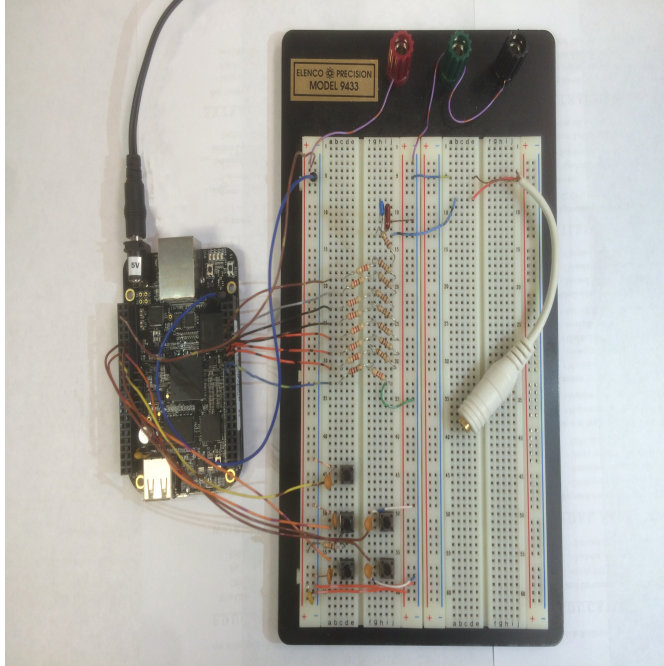
After a digital signal is converted to analog, its amplitude is still quantized. Before it can be outputted as a continuous signal, its amplitude must be smoothed out between the different values. In the frequency domain, this is effectively low-pass filtering the signal. Although there are many different ways to implement a low-pass filter, we decided for our project that the most efficient way would be with a simple RC circuit. Like the R/2R ladder, a basic RC circuit can be implemented very cheaply and on a small scale, since it is composed of only a resistor and a capacitor. Since it is also very simple, our signal can propagate through it very fast. There is one potential problem with the RC (first-order) low-pass filter is that it attenuates more slowly than higher order filters, therefore not completely removing out higher frequencies. However, since the human ear cannot hear above 20,000 Hz, this is not a problem for our implementation. We just require a filter good enough to remove the majority of the higher frequencies so as to not waste power and potentially damage the speakers, as well as to smooth out the different quantized levels in the time domain. A simple RC low-pass filter serves that purpose.

Although there are other ways to implement a D/A converter, for digital signals of 8 bits or less the R/2R ladder is one of the best options. Some of the advantages of the R/2R ladder is that it is composed of only resistors of two different values, allowing it to be very easily implemented on a small-scale at a low cost (small resistors of specific values are can be cheaply produced). Some potential drawbacks of using an 2/2R ladder is that with longer ladders, the cumulative capacitance of the system could potentially delay the transmission of the signal as it is converted from digital to analog. However, since we are using 8 bit signals, there are only 8 rungs on our ladder, so there is no significant delay time due to the capacitance. Another potential problem with R/2R ladders in general is that with the more significant bits, the precision of the resistors are increasingly important. A small fluctuation in these resistors can completely overwhelm the output values of the smaller bits. Fortunately for us, since there are only 8 rungs in the ladder, only so much precision is required for our resistors in the most significant bit.

## Implementation

Goes over the implementation of our project.

## Overall Design



This is an image of our actual implementation of this project. As you can see, the BeagleBone Black on the left is not attached to any supporting computer. We have the data stored directly on the board, and have 5 volts coming in to power it from a power outlet. This power lets the BeagleBone Black perform to the best of its capabilities when generating its bit stream. So 8 bits of digital output streams leave the general purpose I/O and enter the R/2R ladder. The system takes in these 8 digital outputs and generates a voltage that is the weighted sum of each bit. The signal continues through the design, and into the RC low pass filter where the high frequency copies are filtered out. Finally, the signal goes through the audio jack, into speakers, where it gets amplified and played as sound. Also not that we incorporated input buttons into our design for an educational, live demonstration of the process.

## Full digital Signal

In order to construct the full digital signal that is outputted by the BeagleBone Black, first we had to store the original digital data. We took audio files, turned them into a RAW format where they are unsigned bits so that they are easily handled. The BeagleBone Black outputs 8 new bits 44,100 times per second. These 8 output bits are changed to a 0 or a 1 based upon which audio file we read. As covered earlier, this yields 256 total voltage levels of output.

The program we wrote outputs the digital realization of several types of functions as well as 8 bit unsigned audio files to select pins on the BeagleBone Black. It is controlled by polling 5 input pins connected to tactile switches that cycle function, frequency (if applicable) and play/stop. This program relies on the Beagle Bone Black GPIO library.

First in our code, we want to establish pins, functions and songs.

### Definitions

```
#define PI 3.14159265
#define PORT 8
```

```

#define PORT2 9

//output pins
#define PIN0 19 //least significant
bit
#define PIN1 17
#define PIN2 16
#define PIN3 15
#define PIN4 14
#define PIN5 13
#define PIN6 12
#define PIN7 11 //most significant
bit

//input pins
#define PIN8 15
//play/stop button pin
#define PIN9 12
//increase frequency pin
#define PIN10 11
//decrease frequency pin
#define PIN11 14
//next function pin
#define PIN12 13
//last function pin

#define NUMFUNCTIONS 5 //used
to check if function increment should roll over

#define SONG0 "Beethoven5.raw" //define
the audio file names
#define SONG1 "TomSawyer.raw"
#define SONG2 "HarderBetterFaster.raw"

```

Now, we want to establish core functions for both the conversion process and the live educational demonstration.

### Functions

```

//Function: check_low
//Checks the state of a button/pin and waits for end of button press
//Input:
// pin: the number of the pin to check
//Returns:
// 1 if button is pressed/pin is low after waiting for the button to be
// released/pin to return to high
// 0 if button is not pressed/pin is high
int check_low(int pin){
    if (is_low(PORT2, pin)){
        while(is_low(PORT2, pin)){
            iolib_delay_ms(1);
        }
        return 1;
    }
    else return 0;
}

//Function: stop
//Polls the stop pin/button
//Returns:
//1 if button is pressed/pin is low after waiting for the button to be
// released/pin to return to high
//0 if button is not pressed/pin is high
int stop(){
    return check_low(PIN8);
}

//Function: play
//Updates the 8 output pins with a new value
//Input:
// out: the 8 bit value to output
void play(int out){
    if (out & 0x1) pin_high(PORT, PIN0);

```

```

        else pin_low(PORT, PIN0);
        if ((out & 0x2) >> 1) pin_high(PORT, PIN1);
        else pin_low(PORT, PIN1);
        if ((out & 0x4) >> 2) pin_high(PORT, PIN2);
        else pin_low(PORT, PIN2);
        if ((out & 0x8) >> 3) pin_high(PORT, PIN3);
        else pin_low(PORT, PIN3);
        if ((out & 0x10) >> 4) pin_high(PORT, PIN4);
        else pin_low(PORT, PIN4);
        if ((out & 0x20) >> 5) pin_high(PORT, PIN5);
        else pin_low(PORT, PIN5);
        if ((out & 0x40) >> 6) pin_high(PORT, PIN6);
        else pin_low(PORT, PIN6);
        if ((out & 0x80) >> 7) pin_high(PORT, PIN7);
        else pin_low(PORT, PIN7);
    }
    //Function: playSine
    //Calculates the next value of sine for the given frequency and calls
    //play
    //Input:
    //    freq: the frequency of the desired sine wave
    void playSine(int freq){
        int t = 0;
        int out = 0;
        while(!stop()){
            out = 128*sin(t++*freq*PI/22050)+128;
            //calculate the next value to output and increment discrete time
            play(out);
            clock_gettime(CLOCK_REALTIME, and ttime);
            //get the time
            ttime.tv_nsec = 0;
            //clear the nanoseconds but not seconds so that we dont overflow;
            clock_settime(CLOCK_REALTIME, and ttime);
            //set the time back
            clock_gettime(CLOCK_REALTIME, and ttime);
            //get the time again
            ttime.tv_nsec += 21463;
            //increment by number of nanoseconds we should wait
            while(1){
                //loop to delay next output
                clock_gettime(CLOCK_REALTIME, and curtime);
                if (curtime.tv_nsec > ttime.tv_nsec)
                    break;
            }
        }
    }

    //Function: playTriangle
    //Calculates and outputs the next value of a triangle wave for the given
    //frequency
    //Input:
    //    freq: the frequency of the triangle wave
    void playTriangle(int freq){
        int out = 0;
        int t = 0;
        while(!stop()){
            out = (256/((11025/freq))*((11025/freq) - abs(t++ % (2*11025/freq) -
            (11025/freq)))); //calculate next value and increment discrete time
            play(out);
            clock_gettime(CLOCK_REALTIME, and ttime);
            //get the time
            ttime.tv_nsec = 0;
            //clear
            the nanoseconds but not seconds so that we dont overflow;
            clock_settime(CLOCK_REALTIME, and ttime);
            //set
            the time back
            clock_gettime(CLOCK_REALTIME, and ttime);
            //get
            the time again
            ttime.tv_nsec += 21463;
            //increment by number of nanoseconds we should wait
            while(1){
                //loop
                to delay next output
                clock_gettime(CLOCK_REALTIME, and curtime);

```

```

        if (curtime.tv_nsec > ttime.tv_nsec)
            break;
    }

}

//Function: playSquare
//Updates output to maximum and minimum values at the given frequency to
//create a square wave
//Input:
//    freq: the frequency of the square wave
void playSquare(int freq){
    int out = 0;
    int delay;
    while(!stop()){
        out ^= 0xFFFF;
//exclusive or all the bits to make a square wave
        play(out);
        delay = 1000000000/freq/2;
//calculate how long to delay to match frequency
        clock_gettime(CLOCK_REALTIME, and ttime);
//get the time
        ttime.tv_nsec = 0; //clear
the nanoseconds but not $
        clock_settime(CLOCK_REALTIME, and ttime); //set
the time back
        clock_gettime(CLOCK_REALTIME, and ttime); //get
the time again
        ttime.tv_nsec += delay;
//increment by number of nanosec$
        while(1){ //loop
to delay next output
            clock_gettime(CLOCK_REALTIME, and curtime);
            if (curtime.tv_nsec > ttime.tv_nsec)
                break;
        }
    }
}

//Function: playSong0
//Outputs a raw 8 bit unsigned audio file as defined above
void playSong0(void){
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) == NULL) perror("getcwd() error"); //Get the
current working directory
    char* path;
    path = malloc(strlen(cwd) + strlen(SONG0)+2);
//Append the defined file name to the path
    strcpy(path, cwd);
    char temp[2] = "/";
    strcat(path, temp);
    strcat(path, SONG0);
    FILE *fp;
    fp = fopen(path, "r");
    if (fp == 0) perror("error opening file, is the name correct?");
    int out = fgetc(fp);
    while((out != EOF) and !stop()){
//play each of the stored values until end of file or the stop button is pressed
        play(out);
        clock_gettime(CLOCK_REALTIME, and ttime);
        ttime.tv_nsec = 0;
        clock_settime(CLOCK_REALTIME, and ttime);
        clock_gettime(CLOCK_REALTIME, and ttime);
        ttime.tv_nsec += DELAY;
        while(1){
            clock_gettime(CLOCK_REALTIME, and curtime);
            if (curtime.tv_nsec > ttime.tv_nsec)
                break;
        }
        out = fgetc(fp);
    }
}

```

```

    }
    fclose(fp);
}

//Function: playSong1
//Outputs a raw 8 bit unsigned audio file as defined above
void playSong1(void){
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd))== NULL) perror("getcwd() error");
    char* path;
    path = malloc(strlen(cwd) + strlen(SONG1)+2);
    strcpy(path, cwd);
    char temp[2] = "/";
    strcat(path, temp);
    strcat(path, SONG1);
    FILE *fp;
    fp = fopen(path, "r");
    if (fp == 0) perror("error opening file, is the name correct?");
    int out = fgetc(fp);
    while((out != EOF) and and !stop()){
        play(out);
        clock_gettime(CLOCK_REALTIME, and ttime);
        ttime.tv_nsec = 0;
        clock_settime(CLOCK_REALTIME, and ttime);
        clock_gettime(CLOCK_REALTIME, and ttime);
        ttime.tv_nsec += DELAY;
        while(1){
            clock_gettime(CLOCK_REALTIME, and curtime);
            if (curtime.tv_nsec > ttime.tv_nsec)
                break;
        }
        out = fgetc(fp);
    }
    fclose(fp);
}

//Function: playSong2
//Outputs a raw 8 bit unsigned audio file as defined above
void playSong2(void){
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd))== NULL) perror("getcwd() error");
    char* path;
    path = malloc(strlen(cwd) + strlen(SONG2)+2);
    strcpy(path, cwd);
    char temp[2] = "/";
    strcat(path, temp);
    strcat(path, SONG2);
    FILE *fp;
    fp = fopen(path, "r");
    if (fp == 0) perror("error opening file, is the name correct?");
    int out = fgetc(fp);
    while((out != EOF) and and !stop()){
        play(out);
        clock_gettime(CLOCK_REALTIME, and ttime);
        ttime.tv_nsec = 0;
        clock_settime(CLOCK_REALTIME, and ttime);
        clock_gettime(CLOCK_REALTIME, and ttime);
        ttime.tv_nsec += DELAY;
        while(1){
            clock_gettime(CLOCK_REALTIME, and curtime);
            if (curtime.tv_nsec > ttime.tv_nsec)
                break;
        }
        out = fgetc(fp);
    }
    fclose(fp);
}

```

Finally, in a portion of the main section of our code, we need to call these established functions in order to update pins and respond to button presses.

**Main**



```

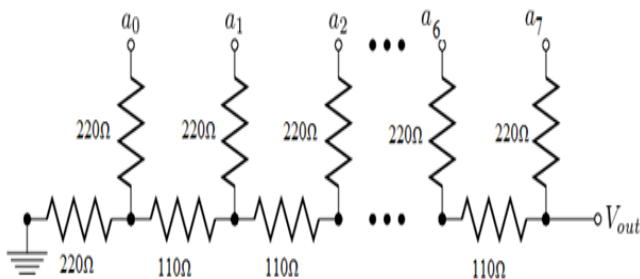
//Function: main
//Polls buttons for input and calls respective functions
int main(void){
    int ret[13];
    iolib_init();
//initialize i/o library
    //intialize the pins to output and input
    ret[0] = iolib_setdir(PORT, PIN0, DIR_OUT);
    ret[1] = iolib_setdir(PORT, PIN1, DIR_OUT);
    ret[2] = iolib_setdir(PORT, PIN2, DIR_OUT);
    ret[3] = iolib_setdir(PORT, PIN3, DIR_OUT);
    ret[4] = iolib_setdir(PORT, PIN4, DIR_OUT);
    ret[5] = iolib_setdir(PORT, PIN5, DIR_OUT);
    ret[6] = iolib_setdir(PORT, PIN6, DIR_OUT);
    ret[7] = iolib_setdir(PORT, PIN7, DIR_OUT);

    ret[8] = iolib_setdir(PORT2, PIN8, DIR_IN);
    ret[9] = iolib_setdir(PORT2, PIN9, DIR_IN);
    ret[10] = iolib_setdir(PORT2, PIN10, DIR_IN);
    ret[11] = iolib_setdir(PORT2, PIN11, DIR_IN);
    ret[12] = iolib_setdir(PORT2, PIN12, DIR_IN);
    int function = 3;
    int frequency = 500;
    while (1){
//poll for user input from buttons
        //check each pin for update
        if(check_low(PIN8)){
            printf("PIN8 pressed");
            if (function == 0){
            else if (function == 4){
                playSong1();
            }
            else if (function == 5){
                playSong2();
            }
        }
        else if (check_low(PIN9)){
//increase frequency by 100 Hz
            printf("PIN9 pressed");
            frequency += 100;
            if (frequency and gt; 20000){
                frequency = 20000;
            }
        }
    }
    iolib_free();
    return (0);
}

```

## R/2R Ladder

Our implementation of the R/2R ladder consists of resistors of  $110\Omega$  and  $220\Omega$ , cascaded together to form eight different rungs, each fed by an output bit from the BeagleBone Black, as shown by the diagram.

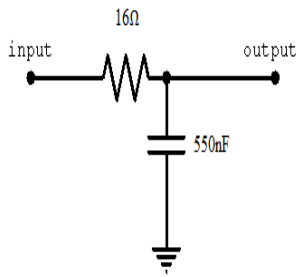


In this configuration, the output voltage,  $V_{out}$ , can be calculated by the following formula:  $V_{out} = V_{ref} * (A/2^8)$  where  $V_{ref}$  is the 3.3V positive voltage of the BeagleBone Black and  $A$  is the binary value currently outputted by the ai's ( $0 \leq A \leq 255$ ). This is because the lower significant bits are scaled-down by the  $110\Omega$  resistors between them and  $V_{out}$ ; each rung between a bit and the output voltage scale the bit by a factor of  $1/2$  before it reaches the output. With this system, we can use change the values of the 8 bits to output any one of 256 different voltages, scaled by 3.3V.

One thing we considered when building the R/2R ladder is the tolerance of the resistors used in the last few rungs. Since the value of the most significant bit is hundreds of times more significant than the value of the least significant bit, we took extra precautions to select and use only the resistors with very tight tolerances for the last few rungs, especially the last rung. If we had not done so and the values of the last two resistors were off by a significant percentage, then our output signal would be considerably distorted along the amplitude axis; the highest possible output value might actually be higher or lower than expected and vice versa for the lowest possible output value. However, during our tests of the system, the oscilloscope showed no significant distortion of the output voltages, and our final output signal played as expected.

## RC Low Pass Filter

When we implemented the RC low-pass filter, we chose the values of the resistor and capacitor to obtain a cutoff frequency (at which the gain is 0.707) of approximately 20,000 Hz. We decided to use a capacitor of 550nF and a resistor of  $16\Omega$ , as shown below.



The magnitude of the transfer function given by this system is:

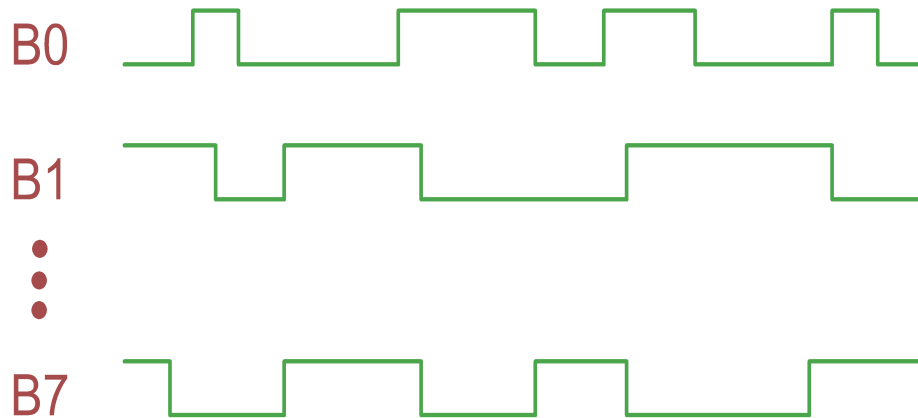
$$|H(j2\pi f)| = \frac{1}{\sqrt{1 + (2\pi fRC)^2}} = \frac{1}{\sqrt{1 + (2\pi f(16)(550 * 10^{-9}))^2}}$$

Given this transfer function, we can calculate the cutoff frequency to be 18,086 Hz. We could not find resistors and capacitors to have a cutoff frequency of exactly 20,000 Hz, but for audible sounds, we decided that 18,000 Hz will be enough to serve our purpose.

## Results

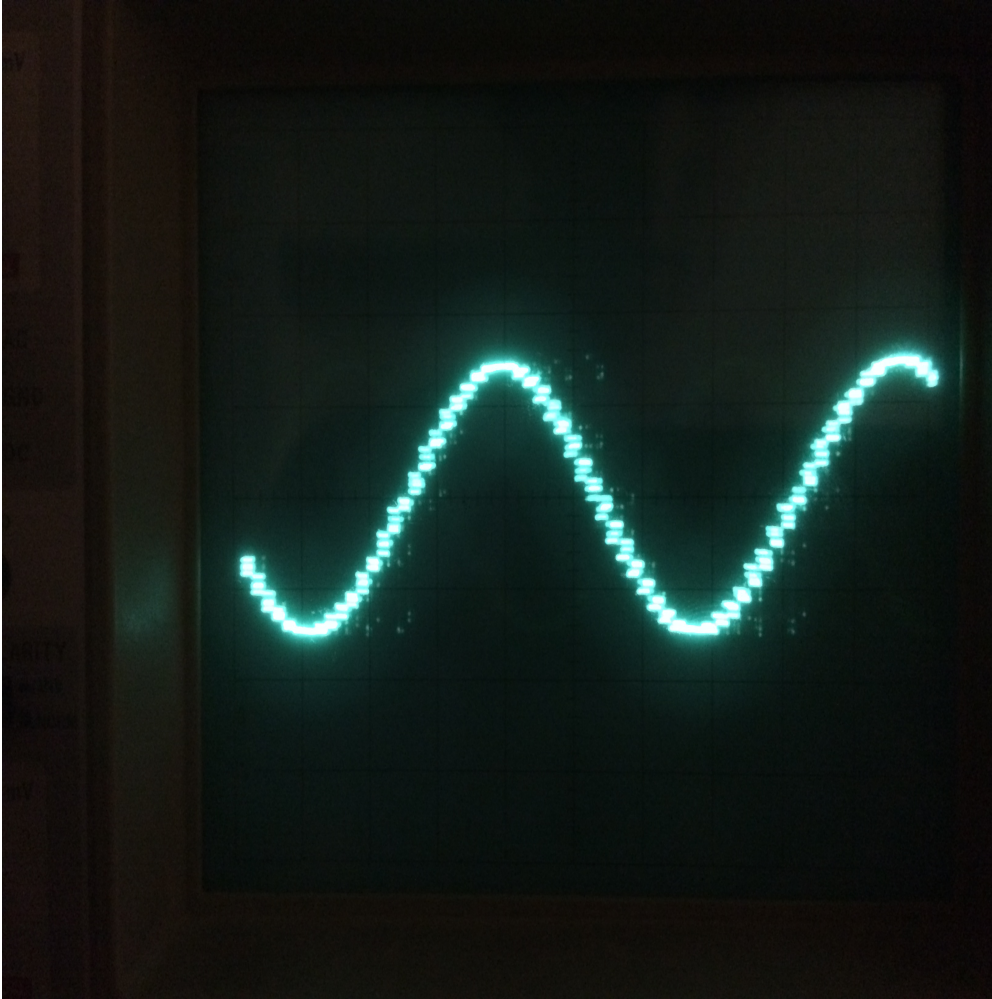
Results of the implementation of our project.

### Full Digital Signal Output



The full digital signal output exists in the digital realm, but it is a bit stream that gets 8 new bits 44,100 times each second. As discussed earlier, the exact nature of this bit stream is based on the original audio file, but there are a total of 256 different output voltage levels.

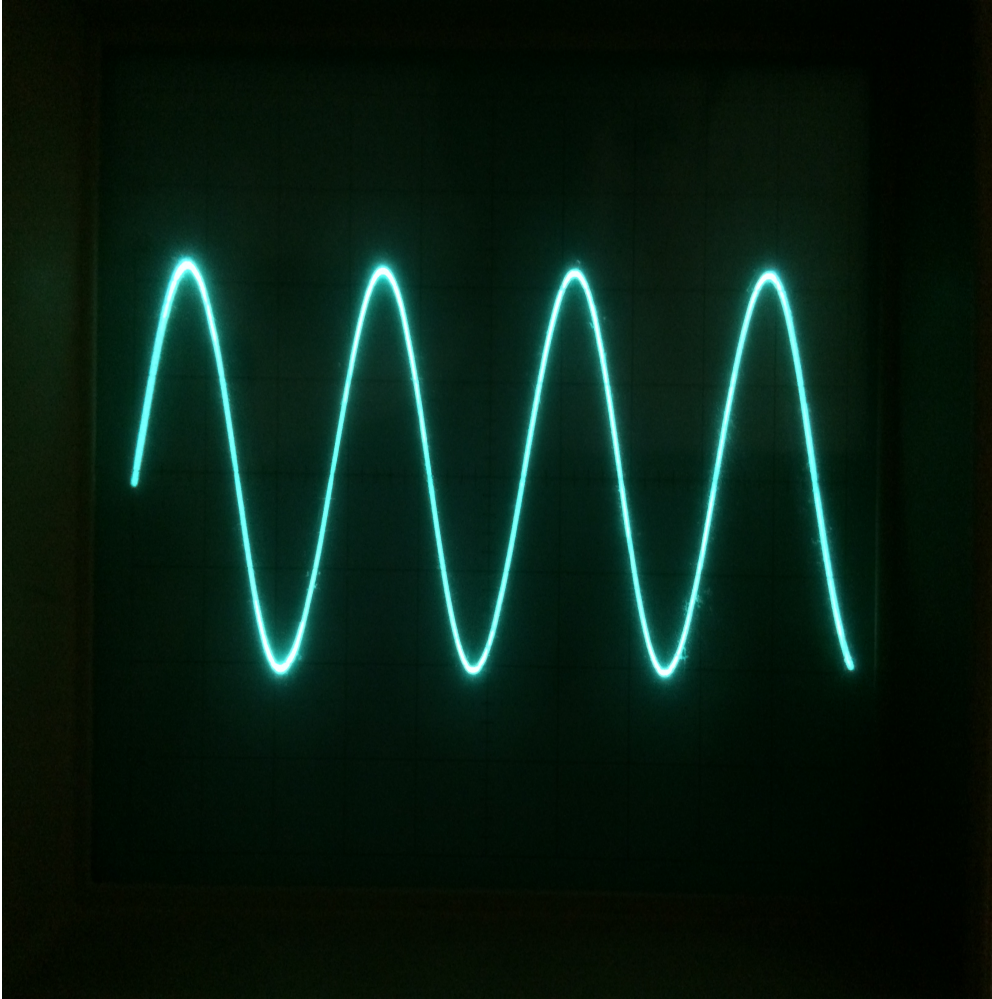
### R/2R Ladder Output



The full digital signal enters the R/2R Ladder and the voltage at output becomes the weighted sum of each bit. The 256 quantization levels make the signal appear blocky and really shows that higher frequencies have been introduced into the signal.

The full digital signal enters the R/2R Ladder and the voltage at output becomes the weighted sum of each bit. The 256 quantization levels make the signal appear blocky and really shows that higher frequencies have been introduced into the signal.

## **Audio Output**



## **Kernel Multitasker**

In our final results, we noticed that the final audio output had significant random delays. This is due to the kernel multitasker on the BeagleBone Black. Since it runs Linux, and Linux is not a realtime system, the processor will choose to run a background process that is not our code from time to time. We used the BBIOlib which directly addresses the general purpose I/Os which avoid many of the Linux performance issues, but not this one. We wanted to collect data on our systems signal to noise ratio, but because of this significant and random delay, it would make our data significantly off. The final sound is pretty clear when coming through the speakers, other than the intermittent delay. However, we could calculate the signal to quantization noise ratio (SQNR) which was around 48 dB.

## Conclusions

Conclusions on the results of our implementation.

## Performance on Characteristics

Overall, we thought that our implementation of this D/A converter was a great success. As far as the characteristics of a D/A converter, we succeeded with several of them which in turn causes the others to be impaired. The power consumption of our system was low because we filtered out the power hungry high frequencies and implemented it completely passively. This, in turn, impacts the size of our design. It was small due the small number of components. The cost of our system was very low, just a handful of simple components and low power consumption. Finally, for speed, a 44.1kHz reproduction rate is all that one desires when reproducing audio signals like we were. Our implementation had the shortcomings of accuracy and resolution. The system was inaccurate because of the Linux kernel multitasker delaying the signal at random intervals. Our choice of a low bit count in conjunction with an R/2R ladder forces the resolution to be low and introduces quantization error.

## Future Improvements

There are future improvements that can be made to make this system even better. In order to tackle the Linux kernel multitasker issue, we cannot just set the priority of our designated process. Other background processes are forced to run, but we have the option to install a real-time kernel. This is a fairly substantial task and exists outside the scope of ELEC 301. However, the realtime kernel would be more accurate because it would be able to run our process exclusively and in realtime. We also could perform up-sampling on the fly which is interpolation that approximates what sampling at a higher rate would do. We discussed using upsampling in conjunction with splines in ELEC 301, but the implementation would be rather difficult. Finally, in order improve our resolution, we could increase the number of bits that our BeagleBone Black outputs. However, the more bits that we use, the lower the frequency that we can transmit them at is because the BeagleBone Black has limited processing speed. Also, if we upgrade to a

16-bit system for instance, we would need to choose a design other than the R/2R implementation because as discussed earlier, the capacitance and tolerance of the system begin to matter.



## **The Team**

Team Vu-Tam Clan, Dr. Simar, and Dr. Malloy all were integral to this project.

## **The Vu-Tam Clan**

The members of our team are Eric Miller, Tyler Taldone, Tam Vu, and Paul Rockaway. We are all Electrical and Computer Engineers at Rice University. Tam and Tyler are also both pursuing a double major in Mathematics. We have some background in analog design, and deep experience in digital implementations. Eric Miller worked on Verilog projects and also poster printing over the summer for both the EE Department and the Digital Media Commons. Tyler Taldone has experience in PCB design and efficiency. Tam and Paul both worked in a neuroengineering design lab and have significant design experiences.

## **Dr. Ray Simar**

Dr. Simar was our mentor for this project and gave us guidance and support. He is a Professor in the Practice of Digital Signal Processing Architecture and teaches Computer Architecture. In this class we were given the BeagleBone Black and encouraged to experiment and design with it. He was recently elevated to Senior Member in IEEE.

## **Dr. Derrick Malloy**

Dr. Derrick Malloy provided us a copy of his book, Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux, so that we could better understand the hardware and software on the BeagleBone Black. Also, when trying to solve the Linux multitasker issue, he personally advised us on the different methods we could employ. We are also thankful that he cautioned us away from implementing it because it would have been a substantial task. We also appreciate his praise for our design, saying that it was “very well executed.” We sincerely enjoyed this project and do look forward to making improvements on it in the future. Note that attached to this module is a link to download our poster for this project in pdf format.